Sesión 5: Construcción de Software

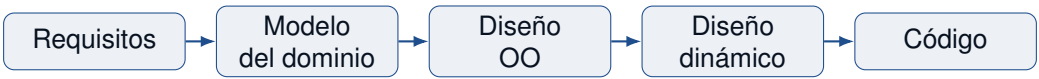
Del Diseño al Código, TDD, Calidad e Integración

Proyecto Integrador: SafeRoute

Sábado 3 de octubre de 2026

¿Dónde estamos en el proyecto?

Durante las sesiones anteriores construimos progresivamente una solución:



Sesiones 2–4

Construimos modelos que responden:

- ¿Qué necesita el usuario?
- ¿Qué objetos existen?
- ¿Quién es responsable?
- ¿Cómo colaboran?

Sesión 5

Ahora debemos responder:

¿Cómo convertimos esas decisiones en software funcional?

Construcción de software

La construcción es la creación detallada de software funcional mediante actividades como:

- codificación;
- verificación;
- pruebas unitarias;
- integración de componentes;
- corrección y refactorización.

Construir software no significa simplemente escribir código.

La construcción debe respetar las decisiones tomadas durante el análisis y el diseño:

Requisitos $\rightarrow$ **Diseño** $\rightarrow$ **Implementación** $\rightarrow$ **Verificación**

Del diseño orientado a objetos al código

El diseño proporciona una guía para estructurar la implementación.

Diseño		**Implementación**
Clase	$\rightarrow$	`class`
Atributo	$\rightarrow$	Campo o propiedad
Operación	$\rightarrow$	Método
Asociación	$\rightarrow$	Referencia entre objetos
Multiplicidad	$\rightarrow$	Colección o relación apropiada
Mensaje	$\rightarrow$	Llamada a una operación

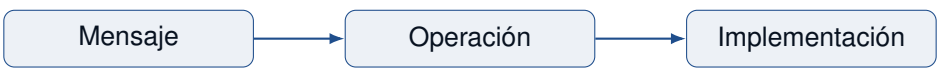
El objetivo no es traducir el UML literalmente, sino implementar las responsabilidades y colaboraciones que el diseño justifica.

Del mensaje del diagrama de secuencia al método

En la sesión anterior vimos mensajes como:

```
envio.agregarSuministro(suministro)
```

Ese mensaje puede conducir a una operación del objeto receptor:



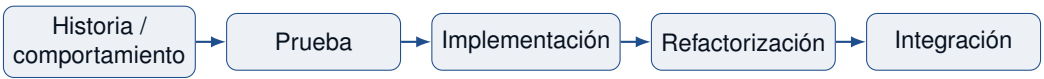
Implica que `Envio` debe tener una responsabilidad relacionada con la gestión de sus suministros.

> La implementación debe ser consecuencia del diseño, no una actividad independiente de él.

Construcción incremental

En lugar de implementar todo el sistema de una sola vez:

Construimos una pequeña parte funcional y verificable.

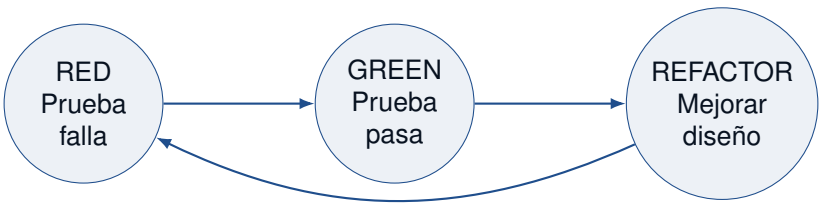


Ventajas:

- Reduce el tamaño de los cambios.
- Permite detectar errores temprano.
- Facilita la revisión del código.
- Produce incrementos funcionales.

Desarrollo guiado por pruebas: TDD

Test-Driven Development propone escribir primero una prueba que exprese el comportamiento esperado.



- **RED:** escribir una prueba que falle.
- **GREEN:** implementar lo mínimo necesario para pasarla.
- **REFACTOR:** mejorar el código sin cambiar el comportamiento.

TDD aplicado a SafeRoute

Regla de negocio:

> Un envío no puede superar la capacidad máxima del vehículo.

Primero expresamos el comportamiento mediante una prueba:



El ciclo permite desarrollar el comportamiento paso a paso, manteniendo una verificación automática.

Anatomía de una prueba unitaria: AAA

Una prueba puede organizarse en tres fases:

Arrange

Preparar el escenario.

- Crear objetos.
- Definir datos.
- Establecer condiciones iniciales.

Act

Ejecutar el comportamiento.

- Invocar el método.
- Ejecutar la operación bajo prueba.

Assert

Verificar el resultado.

- Comparar resultados.
- Verificar estados.
- Verificar errores.

Una prueba debe hacer evidente qué comportamiento se está verificando y qué resultado se espera.

Código limpio: diseñado para ser leído

El código se modifica y se lee constantemente. Por ello, la legibilidad es una propiedad de calidad.

Nombres

- Revelar intención.
- Evitar abreviaturas ambiguas.
- Usar vocabulario del dominio.

Funciones

- Responsabilidad clara.
- Nivel de abstracción coherente.
- Evitar lógica innecesariamente compleja.

Menos claro		Más expresivo
`calcPeso(e)`	$\rightarrow$	`calcularPeso(envio)`
`v`	$\rightarrow$	`vehiculo`
`proc()`	$\rightarrow$	`validarCapacidad()`

Code Smells: síntomas de problemas de diseño

Un **Code Smell** es un síntoma en la estructura del código que puede indicar un problema de diseño.

Duplicación

La misma lógica aparece en varios lugares.

¿Dónde debería vivir esta responsabilidad?

Método gigante

Una operación concentra demasiadas tareas.

¿Qué responsabilidades pueden separarse?

Feature Envy

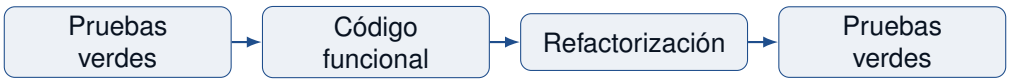
Un método depende excesivamente de otra clase.

¿Está en el objeto correcto?

Los Code Smells no son necesariamente errores:
son señales que justifican una revisión del diseño.

Refactorización: mejorar sin cambiar el comportamiento

La refactorización modifica la estructura interna del código sin cambiar su comportamiento externo observable.

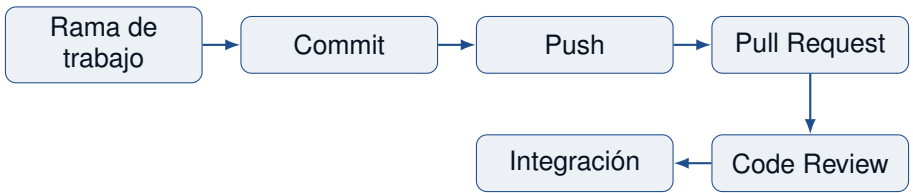


Ejemplos:

- Extraer un método.
- Renombrar una variable o método.
- Eliminar duplicación.
- Dividir una responsabilidad excesiva.
- Mejorar la estructura sin alterar el comportamiento.

Del código individual al trabajo en equipo

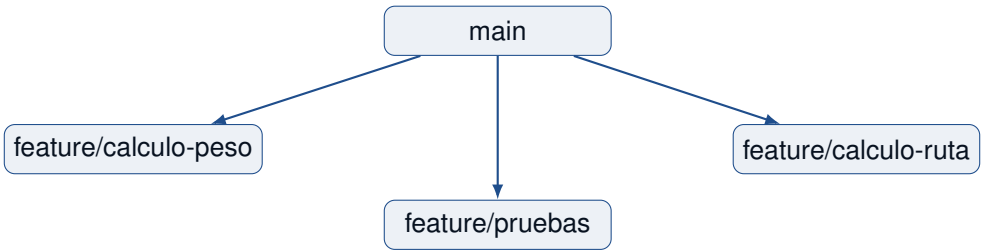
Cuando varias personas construyen el mismo sistema necesitamos controlar los cambios.



La integración deja de ser un evento al final del proyecto y se convierte en una actividad controlada del proceso de desarrollo.

Estrategias de branching

Una estrategia de branching define cómo se organiza el trabajo paralelo en un repositorio. Para SafeRoute podemos trabajar con:



Trabajo aislado → integración controlada → código compartido

Git Flow es una estrategia posible; el principio fundamental es controlar cómo los cambios pasan del trabajo individual a la base compartida.

Commits, Pull Requests y Code Review

Commit

Debe representar un cambio lógico y comprensible.

Ejemplo:

```
feat: calcular peso
total del envío
```

Pull Request

Propone integrar cambios en una rama compartida.

Permite revisar el cambio antes de integrarlo.

Code Review

Otra persona inspecciona el cambio.

Se revisan comportamiento, diseño, pruebas y legibilidad.

El objetivo de una revisión no es encontrar culpables: es mejorar el producto y compartir conocimiento dentro del equipo.

Construcción segura: validar y fallar de forma controlada

Las entradas pueden ser incorrectas, incompletas, inesperadas o potencialmente maliciosas.

Principios:

1. Validar las entradas en las fronteras apropiadas.
2. Mantener invariantes del dominio.
3. Fallar de manera explícita y controlada.
4. No silenciar excepciones.
5. No exponer información técnica innecesaria al usuario.

Invariantes de SafeRoute:

```
capacidadVehiculo > 0
pesoEnvio <=
capacidadVehiculo
```

La seguridad también consiste en impedir que el sistema llegue a estados inválidos.

SafeRoute: de diseño a código y prueba

Consideremos una responsabilidad que ya podemos derivar del diseño:

Calcular el peso total de un envío

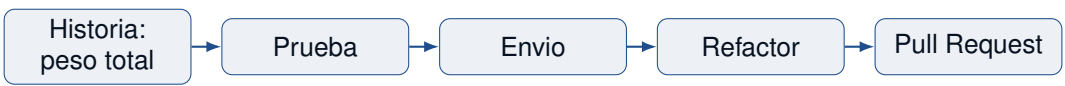


El flujo de trabajo completo:

Diseñar $\rightarrow$ **Implementar** $\rightarrow$ **Probar** $\rightarrow$ **Refactorizar** $\rightarrow$ **Integrar**

SafeRoute: una vertical slice

En lugar de implementar todo el módulo, construiremos una pequeña funcionalidad completa.



Resultado

Al finalizar, el equipo no solamente tendrá código: tendrá una funcionalidad pequeña, probada, versionada y revisada.

Taller: construir una funcionalidad de SafeRoute

Tiempo: 60 minutos

Historia de usuario

Como sistema de distribución, quiero conocer el peso total de un envío para verificar que pueda ser transportado por un vehículo.

Flujo de trabajo:

1. Crear una rama `feature/calculo-peso-envio`.
2. Escribir una prueba para el comportamiento esperado.
3. Ejecutar la prueba y obtener **RED**.
4. Implementar lo mínimo necesario.
5. Obtener **GREEN**.
6. Refactorizar manteniendo las pruebas verdes.
7. Crear un commit atómico.
8. Abrir un Pull Request.
9. Realizar una revisión por pares.

Taller: criterios de entrega

Cada equipo deberá demostrar:

1. **Repositorio:** proyecto SafeRoute disponible en Git.
2. **Rama de trabajo:** una rama `feature/*` para la funcionalidad.
3. **Código:** implementación de la responsabilidad de cálculo.
4. **Pruebas:** al menos una prueba automatizada que demuestre el comportamiento.
5. **TDD:** evidencia del ciclo Red $\rightarrow$ Green $\rightarrow$ Refactor.
6. **Commit:** cambio lógico con mensaje descriptivo.
7. **Pull Request:** solicitud de integración revisada por otro integrante.

Preguntas para revisar el trabajo

Antes de integrar el cambio:

1. ¿La implementación corresponde al diseño realizado anteriormente?
2. ¿La responsabilidad está asignada al objeto adecuado?
3. ¿La prueba describe un comportamiento real del dominio?
4. ¿La prueba falla antes de implementar?
5. ¿La implementación contiene solamente lo necesario?
6. ¿El código puede leerse y entenderse fácilmente?
7. ¿Existe duplicación o una responsabilidad excesiva?
8. ¿La refactorización mantiene las pruebas verdes?
9. ¿El commit representa un cambio lógico?
10. ¿La revisión de código aporta alguna mejora concreta?

Cierre: del diseño al software

Sesiones 3–4

¿Quién es responsable de qué?

$\rightarrow$

Sesión 5

¿Cómo convertimos esas decisiones en código funcional, probado y mantenible?

$\rightarrow$

Construir $\rightarrow$ Probar $\rightarrow$ Mejorar $\rightarrow$ Integrar

El código es la implementación de decisiones de ingeniería, no el punto de partida del proyecto.